# A buffer-sizing Algorithm for Networks on Chip using TDMA and credit-based end-to-end Flow Control

Martijn Coenen[1]
[1]Philips Research
Eindhoven, The Netherlands
martijn.coenen@philips.com

Srinivasan Murali[2]
[2]CSL, Stanford University
Stanford, USA
smurali@stanford.edu

Andrei Rădulescu[1] &
Kees Goossens[1]
andrei.radulescu@philips.com
kees.goossens@philips.com

Giovanni De Micheli[3]
[3]LSI, EPFL
Switzerland
giovanni.demicheli@epfl.ch

## ABSTRACT

When designing a System-on-Chip (SoC) using a Network-on-Chip (NoC), silicon area and power consumption are two key elements to optimize. A dominant part of the NoC area and power consumption is due to the buffers in the Network Interfaces (NIs) needed to decouple computation from communication. Having such a decoupling prevents stalling of IP blocks due to the communication interconnect. The size of these buffers is especially important in real-time systems, as there they should be big enough to obtain predictable performance. To ensure that buffers do not overflow, end-to-end flow-control is needed. One form of end-to-end flow-control used in NoCs is credit-based flow-control. This form places additional requirements on the buffer sizes, because the flow-control delays need to be taken into account. In this work, we present an algorithm to find the minimal decoupling buffer sizes for a NoC using TDMA and credit-based end-to-end flow-control, subject to the performance constraints of the applications running on the SoC. Our experiments show that our method results in a 84% reduction of the total NoC buffer area when compared to the state-of-the art buffer-sizing methods. Moreover, our method has a low run-time complexity, producing results in the order of minutes for our experiments, enabling quick design cycles for large SoC designs. Finally, our method can take into account multiple usecases running on the same SoC.

**Categories and Subject Descriptors:** B.4.3 Input / Output and Data Communications: Interconnections

**General Terms:** Algorithms, Verification

**Keywords:** Systems-on-Chip, Networks-on-Chip, Area, Buffers


*CODES+ISSS'06,* October 22–25, 2006, Seoul, Korea.


## 1. INTRODUCTION

To effectively tackle the increasing design complexity of SoCs, the computation architecture needs to be decoupled from the communication architecture [16]. By such decoupling, the computation and the communication architectures can be designed independently, thereby speeding up the entire design process and hence reducing the time-to-market of SoCs. NoCs can offer such decoupling with decoupling buffers between the computational blocks and the communication blocks, thereby hiding the differences between the operating speeds and burstiness of the cores and the NoC. This allows the cores to execute their transactions without noticing the presence or impact of an interconnect, for example they will not stall if the NoC is busy with another core.

Methods to find the minimum size of the NoC decoupling buffers for the set of applications that are run on the SoC is an important problem for two reasons. First, the decoupling buffers take up a significant amount of the NoC area and power consumption, thus finding the minimum buffering requirements is key to achieve an efficient NoC implementation. Second, for a predictable system behavior, we need to compute the minimum buffering that still satisfies the application requirements.

Moreover, some NoCs employ credit-based end-to-end flow control mechanisms to provide guaranteed system operation and to remove message-dependent deadlocks in the system [1]. In this case, additional buffering is required to hide the end-to-end latency for the flow control mechanism and to provide full throughput operation. If the buffers are too small, then the throughput and latency are affected and no end-to-end guarantees can be given.

In this paper we address the problem of computing the minimum size of the decoupling buffers of the NoC. We present an application-specific design method for determining the minimal buffer sizes for the Guaranteed Throughput (GT) connections of the Æthereal NoC architecture [15]. We model the application traffic behavior and the network behavior to determine the exact bounds on buffer-sizing. In our method, we also consider the buffering requirements due to the use of credit-based end-to-end flow control.

We apply our method to several SoC designs, which show

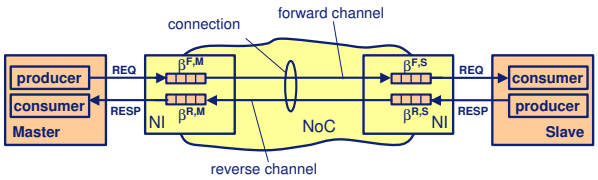


**Figure 1: The buffers for a connection**

that the proposed method leads to a large reduction in the total NoC buffer area (84% on average) and power consumption when compared to an analytical method. Our method has a low run-time complexity and is therefore applicable to complex SoC designs too. The method can be applied for designs with multiple usecases, by taking the maximum required buffer size over all usecases for each buffer. Finally, the method is also integrated into our fully automatic design flow, enabling fast design cycles over a SoC design. Although the algorithmic method is presented for the Æthereal architecture, it can be applied to any NoC for which the behavior of both the IP cores and the network is periodic, such as aSoc [5] and Nostrum [6].

Traditionally, simulation (or trace) based approaches such as [12] are used to compute the buffering requirements in systems. While they provide an optimal bound for the given trace, there is no guarantee that the derived buffer sizes will satisfy different traces. Hence, they cannot be used to build predictable systems. Analytical methods for sizing buffers based on jitter-constrained periodic behavior are known, such as the ones presented in [2, 3]. These methods are usually too pessimistic and can result in larger buffers than required for the design. We quantify this in Section 5. Stochastic approaches based on queuing theory are shown in [7]. Such stochastic models can only approximate the actual traffic characteristics of the application, and hence system behavior cannot be guaranteed.

A general mathematical theory, network calculus [8], has been established to model network behavior.It allows computing bounds on delays and back-logs in networks. The foundations of our proposed algorithmic approach to buffer-sizing are based on the models of network calculus.

*Synchronous Data Flow (SDF)* graphs to model signal processing and multimedia applications have been presented by several researchers [9]. Using SDF models to minimize buffering requirements of processors has been presented in [10]. The use of SDFs to model NoCs has been presented in [11]. The SDF models however assume a uniform data production and consumption to compute the buffering requirements. In NoCs that provide throughput guarantees, the TDMA slots allocated to a traffic stream need not be uniformly spread over time. Thus, SDF models can not model the network in such detail as shown here, and the results are hence less optimal.

## 2. THE Æthereal NOC

The Æthereal NoC architecture uses the notion of connections to represent communication streams between IP cores [15]. Such connections are needed in order to allocate resources such as TDMA slots and buffers for real-time behavior. A connection consists of two channels, a forward channel and a reverse channel (see Fig. 1). On the forward channel requests may be sent from a master to a slave, and on the reverse channel a response can be sent (in case of a read transaction for example). On the forward channel therefore the master is the producer and the slave is a consumer; on the reverse channel these roles are reversed.

Each connection has four buffers in the Network Interfaces (NIs) connecting the IP cores to the network: $\beta^{F,M}$ and $\beta^{F,S}$, indicating the buffers in the master and slave NIs for the forward channel, and $\beta^{R,S}$, $\beta^{R,M}$ for the reverse channel. The buffers in the NI are needed to compensate for the differences in operating speed and burst sizes between the IP cores and the NoC. The reason for each connection to have its own queues is that if connections would share a single queue, dependencies between these connections would arise. If on one connection data is not consumed, it will block the other connections, which in turn could lead to not meeting timing requirements or even deadlock [1].

The NoC provides throughput and latency guarantees by using TDMA [14]. This is implemented by means of slot tables in the NIs, where each slot represents an equal amount of time. Each connection is then assigned a number of slots to match its bandwidth and latency requirements [4]. In every slot a fixed number of words can be sent into the network. The first word is always a packet header, unless the previous slot was occupied by the same connection.

Once data leaves the NI, the NoC guarantees a contention-free path to the target NI [14]. This is achieved by reserving time-aligned slots for each router link. The calculation of the slots and the corresponding contention-free paths through the router network is currently done at design time [4]. Having a contention-free path results in minimal buffers in the routers, because no packet ever has to wait. It is necessary though that data from the routers is always accepted by the NIs, otherwise, if the consumer is slow or does not respond at all, the NI buffers would fill up, finally spilling over in the network. This would break the contention-free routing and guarantees.

In order to avoid this, the Æthereal NoC employs end-to-end flow-control using a credit-based mechanism [17]. Local counters in the NI keep track of the amount of space in buffers of the remote NIs for each connection. Whenever a word leaves a NI, the counter is decremented by one. If the counter reaches zero, the NI is not allowed to send data into the network. Whenever a word is consumed at the consumer NI, a credit is generated and sent back over the network when a time-slot is available for the connection.

Because the credits do not arrive instantaneously, a producer does not immediately know when its data has been consumed. To avoid stalling of the producer if the NI runs out of credits, we must account for the end-to-end flow control in our buffer calculation as well. This affects the buffering calculations both on the forward and the reverse channel. On the forward channel, flow control credits for the reverse channel are sent. On the reverse channel, flow control credits for the forward channel are sent.

The flow control credits are sent in the packet header [14]. As a result, the available payload bandwidth is independent of the credit bandwidth. Within one connection the forward and reverse channels can also be computed independently of eachother. Finally, since each connection has its own buffers in the NIs, the other connections cannot interfere, and we can look at each connection in isolation when considering the size of the buffers in the NI. These independencies re-

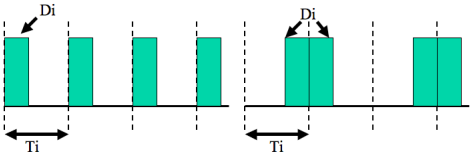


**Figure 2: Periodic (left) and Aperiodic (right)**

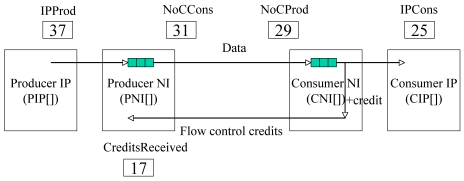


**Figure 3: The flow of data and algorithm variables**

sult in compositionality; connections can be removed and added without affecting the others, thereby making verification easier, and real-time guarantees can easily be maintained. This also allows for simpler and incremental algorithms, such as the buffering algorithm described in this paper, which can calculate the buffering requirements for each connection in isolation.

We describe the buffer-sizing algorithm for the forward and reverse channels. Before this, we need to characterize the application behavior.

# 3. APPLICATION BEHAVIOR

In order to compute the sizes of the decoupling buffers in the NI, we need to characterize the application behavior of the IP cores. We consider three types of production patterns for the IP cores. First, the periodic production pattern, in which a producer produces a burst of fixed size at the same time in each period. An example of the periodic producer pattern with burst size $D_i$ and period $T_i$ is shown on the left in Figure 2.

Second, the aperiodic production pattern, in which a producer produces a burst of fixed size, but the bursts can appear anywhere within the period. Such a model is characteristic of applications with some uncertainty in the time at which the bursts are generated. Most traffic patterns of video processing applications are periodic and bursty in nature [13] and can be modeled by the periodic or the aperiodic production patterns. An example of the aperiodic production pattern is shown on the right in Figure 2.

Finally, the multi-periodic production pattern, in which a producer produces multiple bursts, with different burst sizes and time between the bursts. As an example, in video display systems the bursts of data for each horizontal scan line have a fixed burst size, with a fixed *blanking* (quiet) period between two scan lines. However, after a full set of horizontal scans, there is a bigger blanking period for the vertical scan.

Even if a certain producer behavior does not fit in one of these three production types, several methods can be used to transform it to one of them. Worst-case specifications could be used to compute the period and burst size, for example obtained from analytical estimates or from several experimental runs of the application.

# 4. COMPUTING THE NI BUFFER SIZES

In order to compute the buffer sizes in the NIs, we want to compute the maximum difference between the number of words produced and the number of words consumed at any point in time. We first describe the problem of buffer sizing for a general producer and consumer.

We introduce two arrays *input* and *output* representing the production patterns of a producer and a consumer, where:

*Definition 1. input[t]* has value '1' if the producer produces one value at time $t$, otherwise it has value '0'

*Definition 2. output[t]* has value '1' if the consumer is ready to consume exactly one value at time $t$, otherwise value '0'

Note that *output* describes the availability of the consumer to remove data from the buffer. The buffer may contain data, or may be empty. We define *input.t* to be the number of data items produced in the $[0..t)$:

$$input.t = \sum_{0 \le i < t} input[i] \qquad (1)$$

Similarly we define *output.t* to denote the number of data items consumed in the interval $[0..t)$. Recall that the *output* array only indicates whether the consumer is ready to consume a value, not that a value is actually consumed, because that depends on the availability of data in the buffer. To match the definition of *output.t* with the array *output* we therefore need to include an additional condition to check whether there is currently data in the buffer. Data is available at time-point $j$, when the number of words produced in the interval $[0..j]$ exceeds the number of words consumed in the interval $[0..j)$, $input.(j+1) > output.j$ .

$$output.t = \sum_{0 \le j < t \wedge input.(j+1) > output.j} output[j] \qquad (2)$$

Using Equation (1) and Equation (2) the specification of the minimum required buffer size is the maximum difference between the number of words produced and consumed in an interval of length T:

$$maxbuffer = max_{0 \le t < T}\{input.t - output.t\} \qquad (3)$$

To compute this, producer and consumer traces for an interval can be used to compute the required buffer size in that interval. There is of course no guarantee that a value derived from one trace satisfies the trace of any other, nor that the buffer will be big enough for any data produced outside the interval. In the next section we show how having periodic producer and consumer behavior allows us to compute the buffering requirements for an infinite amount of time.

In the next sections, we first consider the calculation for the NI buffers connected to the producing IPs. Here the IPs are the producer and the NoC is the consumer. Then, we will look at the NI buffers connected to the consuming IPs, where the NoC is the producer and the IPs are consumer. In both cases we only consider the periodic production pattern. In section 4.3, we will show how to extend the algorithm to the other production patterns.

## 4.1 Producer NI buffer calculation

When considering the producer NI buffering for a connection we need to calculate both the size of the forward

master buffer $\beta^{F,M}$ for the master who produces requests on the forward channel, and the size of the reverse slave buffer $\beta^{R,S}$ for the slave who produces response data on the reverse channel. Below we discuss only the forward channel, because the same algorithm is used for the reverse channel.

The algorithm for calculating the producer NI buffer requirements is a straightforward implementation of Equation (3), where two array variables $PIP$ (Producer IP) and $PNI$ (Producer NI) are used to capture the input/output patterns of the IP and NI, and two running variables $IPprod$ and $NoCCons$ are used to store the total number of words produced by the IP and consumed by the NoC respectively, reflecting the left-hand side of Equations (1) and (2) respectively. We compute the consumer NI buffer in the same algorithm, which is explained in detail in the next section. We have visualized the variables in Figure 3, where both the IPs and the NIs are shown. Counter variables such as $IPProd, NoCCons$ are indicated by square boxes, whereas the array variables are shown in the IPs and NIs with brackets behind them.

Now let $T_i$ be the period of the IP block producing data. The consumer in this case is the NI connected to the producer IP, the"producer NI". Because of the slot table, the producer NI is also periodic with a period of $T_o$, equal to the number of slots multiplied by the time duration of one slot. Additionally in each period both the producer IP and producer NI produce/consume a fixed number of words, $D_i$ and $D_o$, respectively. $D_i$ corresponds to the burst of the producer, and $D_o$ is equal to the total number of words allocated for the connection in one slot table revolution.

With this information, we can fill the arrays *'PIP'* (Producer IP) and *'PNI'* (Producer NI). Because of the periodicity the arrays need only be as long as their respective periods. The $PIP$ array is constructed as follows for a periodic producer:

$$PIP[t] = \begin{cases} 1 & \text{if } 0 \leq t < D_i; \\ 0 & \text{if } D_i \leq t < T_i. \end{cases}$$

The $PNI$ array is constructed according to the known slot table allocation for the corresponding NI:

$$PNI[t] = \begin{cases} 1 & \text{if consumer is ready to consume at time t;} \\ 0 & \text{otherwise.} \end{cases}$$

As mentioned in Section 4, we do not have to simulate for an infinite period of time to compute the maximum, because the behavior of both producer and consumer is periodic. The crucial observation is that the behavior of two periodic intervals repeats itself after the least common multiple (lcm) of the two periods. This leads us to calculate the least common multiple of $T_i$ and $T_o$, $T_{lcm}$. After $T_{lcm}$ the producer and consumer will be realigned and the pattern repeats itself.

We present Algorithm 1 for calculating the producer buffer requirements. Lines 1 through 7 initialize. Note that on line 1 we also need to take the period of the consumer into the calculation of the lcm, when considering the consumer NI buffering in the next section. Line 8 shows the time loop which has $lcm$ iterations, in order to compute the maximum difference between producer and consumer in the time-interval $[0..lcm)$. In line 9, the number of words produced until time $n$ is updated by adding $PIP[n\%T_i]$ to $IPProd$.

Line 10 checks whether there are currently words in the producer NI buffer. If there are, in line 11 the $NoCCons$ variable is updated by adding $PNI[n\%T_o]$ (which is one if the NI can send data at that point and zero if not). In line 12, the $NocArriving$ variable is updated to indicate whether a word of data will arrive at the consumer NI $T_{Fwd}$ (the time it takes to traverse the network) time from now. This variable is used to capture the production pattern at the consumer NI, and will be explained in more detail in the next section. Finally in line 14, we see if the current difference between the number of words produced and the number of words consumed is bigger than the current maximum, and if so we replace it in line 15. Lines 17 to 29 are used for the consumer NI buffer calculation and will be explained in the next section.

**Algorithm 1** Calculates the buffer requirement in producer and consumer NIs for a periodic producer and consumer

**Require:** Arrays PIP[0..$T_i$], PNI[0..$T_o$], CNI[0..$T_o$] and CIP[0..$T_c$]

```
 1: lcm ⇐ the least common multiple of T_i, T_o and T_c
 2: IPProd ⇐ 0
 3: NoCCons ⇐ 0
 4: NoCProd ⇐ 0
 5: IPCons ⇐ 0
 6: maxprodbuffer ⇐ 0
 7: maxconsbuffer ⇐ 0
 8: for n = 0 to lcm − 1 do
 9:    IPProd ⇐ IPProd + PIP[n%T_i]
10:    if IPProd − NoCCons > 0 then
11:       NoCCons ⇐ NoCCons + PNI[n%T_o]
12:       NoCArriving[n + T_Fwd] ⇐ PNI[n%T_o]
13:    end if
14:    if IPProd − NoCCons > maxprodbuffer then
15:       maxprodbuffer ⇐ IPProd − NoCCons
16:    end if
17:    NoCProd ⇐ NoCProd + NocArriving[n]
18:    if NoCProd − IPCons > 0 then
19:       IPCons ⇐ IPCons + CIP[n%T_c]
20:       credit ⇐ credit + CIP[n%T_c]
21:    end if
22:    if CNI[n%T_o] ∧ credit > 0 then
23:       creditArriving[n + T_Rev] ⇐ credit
24:       credit ⇐ 0
25:    end if
26:    CreditsReceived ⇐ CreditsReceived
       +creditArriving[n]
27:    if NoCProd − CreditsReceived > maxconsbuffer
       then
28:       maxconsbuffer ⇐ NoCProd − CreditsReceived
29:    end if
30: end for
```

Note that the computation of $IPProd$ is only dependent on the producer itself, but the number of words actually consumed ($NoCCons$) also depends on the availability of flow-control credits. We will deal with flow control in the consumer NI buffering, and make sure that the consumer NI buffer is big enough (and hence enough credits are available to the producer NI) to sustain the required throughput.

## 4.2 Consumer NI buffering

When data is sent from the producer NI, it arrives at the consumer NI after $T_{Fwd}$ time. We assume that we can characterize the behavior of the consumer similar to that of the producer: the consumer is periodic with a period $T_c$ and

consumes bursts of size $D_c$ at the beginning of each period $T_c$. We store this information in the array $CIP$. The total number of words produced and consumed by the NI and the IP are stored in $NoCProd$ and $IPCons$ respectively. Once the consumer consumes data, it produces a credit as shown in Figure 3. The credits are sent back in the first slot that is available for the connection in the consumer NI (the slot table of the consumer NI is contained in the $CNI$ array). Whenever the credits are sent out, they still take the constant network delay $T_{Rev}$ before the producer NI receives them, and there the total number of received credits is stored in a variable $CreditsReceived$. The producer NI hence does not get credits as soon as data has been consumed but only after a delay, and in the meantime a new burst of data from the producer may arrive. Since in the producer NI buffering we assumed there were enough credits available (in order to avoid stalls), we have to make sure the consumer NI buffer is big enough to receive additional words of data while the credits are underway. This leads us to compute the maximum difference between the number of words produced by the network and the number of credits received by the producer NI (instead of the number of words consumed by the consumer).

Lines 17 to 29 of Algorithm 1 are used to calculate the required size of the consumer NI buffer. Line 17 updates the variable $NoCProd$ to reflect the number of words the network has produced until time $n$, by adding the $NoCArriving[n]$ variable to it. This variable is set to 1 whenever the producer NI produces a word $T_{Fwd}$ time earlier in line 12. In line 18 the current filling of the consumer NI buffer is determined. If the difference is greater than zero, $CIP[n\%T_c]$ (indicating if the consumer IP consumes at time $n$) is added to $IPCons$ in line 19. In order to model the credit mechanism, we also add $CIP[n\%T_c]$ to the *credit* variable, which effectively adds a credit whenever the consumer consumes a word. Credits are sent back to the producer NI once there is a slot available, which is checked in line 22. If a slot is available, in line 23 the *creditArriving* array is updated to reflect the arrival of a credit in the producer NI at time $n + T_{Rev}$. Line 24 resets the credit counter after credits have been sent. In line 26 the number of credits that are due to arrive at the producer NI are added to the variable $CreditsReceived$. Finally, the difference between the number of words produced by the network and the number of credits received is calculated and the maximum is stored.

In Section 2 we mentioned that in each slot a packet header needs to be sent before any data, unless a connection was granted the previous slot also. We also mentioned that the flow control credits are stored in a reserved part of the packet header with a size of 5 bits [14], allowing only for a maximum of 32 credits to be sent in each slot. While not shown in Algorithm 1 for simplicity, these issues are taken into account in our implementation.

The producer, NoC and consumer periodic behavior need not be aligned. Therefore we shift two of the three input arrays to and rerun the algorithm for each combination. Thus we compute the minimum buffer size for all possible alignments of the periodic intervals. The time complexity of this algorithm is therefore $O(T_i \times T_o \times (lcm(T_i, T_o, T_c)))$ and thus polynomial.

## 4.3 Other production patterns

In Section 3 we mentioned two other production patterns, the aperiodic production pattern and the multi-periodic production pattern. Both of them can easily be incorporated in the presented algorithm. For the aperiodic pattern, we consider the worst case that can occur: in any time-interval with a length of 2 periods (essentially a sliding window), a maximum of three bursts may arrive. We then model the aperiodic pattern periodically by making the period twice as large and the burst three times as big. This can be reflected in the producer array easily. Whereas this might seem like a lot of overhead, in practice the period of the NI is much smaller than the period of the IPs. During one period of the IP block the NI might have 10 periods in which it consumes data half of the time, thus requiring a much smaller buffer.

For the multi-periodic production pattern, we can consider the highest level at which the production pattern is periodic. For the video display system mentioned in Section 3 this would be the frame period. We can then create a producer array with the length of this period, and then fill in each of the bursts by setting values to 1 whenever a word of data is being transferred in this period.

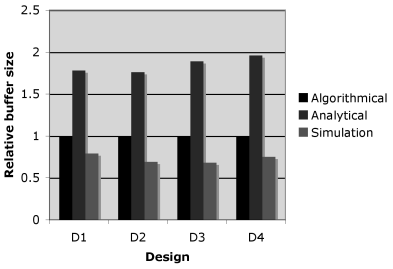


**Figure 4: Relative buffer sizes for various designs**

.

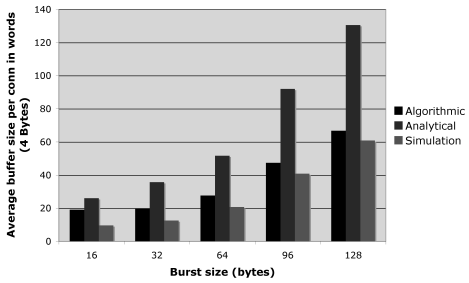


**Figure 5: Effect of burst size on buffer-sizing.**

# 5. RESULTS

We have compared the method described in this paper with the analytical method in [2] and with simulations of the applications. We benchmark the buffer-sizing methods using two different in-house SoC designs, a set-top box SoC (D1) and a multimedia SoC for phones (D2). Each of these designs contains a large number of IP cores (10+) and a large number of connections (20+). In addition to these two designs, we have generated a large amount of synthetic designs divided in two classes: designs with a bottleneck IP core such as a memory (D3) and designs with evenly spread communication (D4). We calculated the buffer sizes for each of these designs, using algorithmic and analytical methods, and compare them to the maximum buffer filling observed during a simulation.

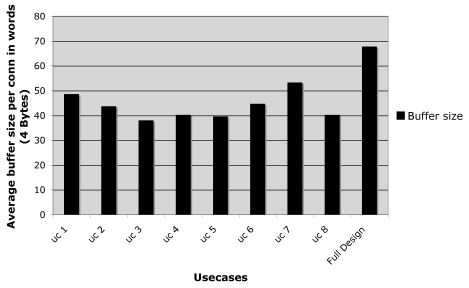


**Figure 6: Buffer size for multiple usecases.**

The result is shown in Figure 4. As can be seen, the algorithmic method performs significantly better than the analytical method, on average the buffers are 84% smaller. The primary reason for this big difference is that the analytical method does not take the periods of the producer and consumer and their alignment into account, but uses a worst-case buffer requirement equal to the sum of the burst sizes of producer and consumer $(D_i + D_o)$ [2].

The maxima achieved during simulation are on average another 38% smaller. This is because the simulation is just a single trace, and the worst case alignment that the algorithm computes does not always occur. Relying on a single trace/simulation for buffer-sizing may result in buffers that are too small. Also, the added consumer buffering to hide the end-to-end flow-control round-trip delay is not always used, since data in the consumer buffer may have been consumed before new data arrives.

Figure 5 shows how the burst size affects the buffer sizes. We have generated 100 synthetic usecases for each burst size, and calculated the buffer sizes for each them. The figure shows the average buffer size per connection for both the analytical and algorithmic methods compared to the maximum observed during simulation. When the burst sizes are very large (128 bytes and more), the algorithmic method only adds 10% to the buffering observed during simulation. This is because increasing the burst size while keeping the bandwidth increases the period too. Typically then within a period only a single burst needs to be buffered, and the flow control credits can be back before the beginning of the next period. When the burst size is small however, the extra buffering in the consumer buffer for flow control is typically large compared to the burst size.

Figure 6 shows the average buffer size per connection for each of the 8 usecases of a single SoC. The NI buffer size for the entire SoC is the maximum buffer size of all usecases, for each connection. The "full design" column shows the result. Because connection buffers may be largest in different use cases, "full design" is larger (by 27%) than the largest usecase (uc7).

The run time is typically in the order of a few minutes, except when a design has connections with low bandwidth requirements (less than a megabyte/second). The large periods result in a big lcm. Optimizing the algorithm to perform better for such connections is part of future work.

# 6. CONCLUSIONS AND FUTURE WORK

NoCs that offer guaranteed services are critical for future SoC design with real-time requirements. In order to provide these services to the IP cores, the NoC must contain decoupling buffers to hide the difference in operating speed between the IP core and the NoC. These buffers are the dominant factor in NoC area and power. Minimizing buffers while still matching the required application behavior is an important problem.

In this paper we presented a novel design method for sizing the decoupling buffers in the NIs of the Æthereal NoC. The method exploits knowledge we have about the behavior of the IP cores and the NoC and can reduce the buffer area in designs on average by 84%, when compared to an analytical worst-case method. The method also takes into account the complicating effects of end-to-end credit-based flow-control on the required buffer sizes. The method is fast for all applications considered, and supports a wide variety of application behavior. Finally, the method can take into account multiple usecases for a single SoC design.

Future work includes research on the the effect of good alignment between the IP cores and the NoC, therefore reducing the buffering requirements even further.

# 7. REFERENCES


[1] A. Hansson *et al.*,Analysis of Message-Dependent Deadlock in Network-Based Systems on Chip. In *Philips Research Technical Note 2006/00230*

[2] O. P. Gangwal *et al.*,Building predictable systems on chip: An analysis of guaranteed communication in the Æthereal network on chip. In P. van der Stok, editor, *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, volume 3 of *Philips Research Book Series*, chapter 1, pages 1–36. Springer, 2005.

[3] C. Hamann. On the quantitative specification of jitter constrained periodic streams. In *Proc. MASCOTS '97*, page 171, Washington, DC, USA, 1997. IEEE Computer Society.

[4] A. Hansson *et al.*,A unified approach to constrained mapping and routing on network-on-chip architectures. In *Proc. CODES+ISSS*, pages 75–80, Sept. 2005.

[5] J. Liang *et al.*, aSOC: A scalable, single-chip communications architecture. In *Proc. PACT*, 2000.

[6] M. Millberg *et al.*,Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proc. DATE*, 2004.

[7] D. Gross, C. Harris Fundamentals of Queueing Theory *Wiley-Interscience*, 1998.

[8] J. Boudec, P. Thiran Network Calculus : A Theory of Deterministic Queuing Systems for the Internet *Lecture Notes in Computer Science, Springer*, 2001.

[9] S. Bhattacharyya *et al.*,Software Synthesis from Dataflow Graphs *The International Series in Engineering and Computer Science, Springer*, 1996.

[10] M. Geilen, T. Basten, and S. Stuijk Minimising buffer requirements of synchronous dataflow graphs with model checking In *Proc. DAC*, 2005.

[11] P. Poplavko *et al.*,Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip In *Proc. International conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[12] S. Murali and G. De Micheli. An application-specific design methodology for STbus crossbar generation. In *Proc. DATE*, 2005.

[13] M. Krunzt, R. Sass, and H. Hughes Statistical characterisitics and multiplexing of MPEG streams In *Proc. Conference of the IEEE Computer and Communications Societies*, 1995.

[14] A. Rădulescu *et al.*,An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4–17, Jan. 2005.

[15] K. Goossens. *et al.*, The Æthereal Network on Chip: Concepts, Architectures, and Implementations. In *IEEE Design and Test of Computers*, 22(5):21–31, 2005.

[16] M. Sgroi *et al.*,Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proc. DAC*, pages 667–672, June 2001.

[17] I. Cidon and K. Goossens. Network and transport layers in networks on chip. In G. De Micheli and L. Benini, editors, *Networks on Chips: Technology and Tools*, The MK Series in SoS, chapter 5, pages 147–202. Morgan Kaufmann, July 2006.